TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Arvutiteaduse instituut

ITV70LT

Maria Kohtla 132387IAPM

GESTURE EVALUATION FOR LEAP MOTION

Master thesis

Jaagup Irve

Master of sciences

Software engineer

Tallinn 2015

Autorideklaratsioon

Deklareerin, et käesolev lõputöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud.

…….. …….

(kuupäev) (lõputöö kaitsja allkiri)

Abstract

The goal of this thesis was to create a simple yet highly functional application for Leap Motion with a practical purpose. That goal led to the idea of the Japanese *hiragana* learning tool.

Application was written in Python using Pygame module for mainly managing UI and keyboard events. For motion control the Leap Motion was used with its software, which provides a lot of different tools for managing its input.

For the need of this application the stroke detection method was developed. Main parts of the method are feature point extraction and the stroke comparison.

The work also gives an overview of the application, including descriptions for the storage method, used to store the correct stroke patterns, also the Leap Motion controls, and different visual elements used.

Annotatsioon

Antud töö eesmärgiks oli luua lihtne, kuid efektiivne ja praktiline rakendus Leap Motioni abil. See viis ideeni luua Jaapani *hiragana* kirjutasimise õppevahend.

Rakendus on kirjutatud Pythonis kasutades Pygame moodulit, põhiliselt kasutajaliidese ja klaviatuuri sündmuste haldamiseks. Käte liikumise tuvastamiseks kasutati Leap Motionit, mille tarvara pakub palju erinevaid meetodeid selle sisendi haldamiseks.

Rakenduse tarvis sai eraldi loodud ka joonsümboli tuvastamise meetod. Antud meetodi põhiosadeks on joonsümboli põhipunktide tuletamine ja joonsümbolite võrdlus.

Lisaks sisaldab töö ka rakenduse ülevaadet, sealhulgas kirjeldusi nii korrektsete sümbolite hoiustamisest, Leap Motioni funktsioonidest ning ka erinevatest kasutatud visuaalsetest elementidest.

Table of Contents

1 Introduction

In almost every science fiction movie, book or series we have ever seen, protagonists use powerful motion control tools with just a wave of their hand. We have been captivated by the notion of those simple, natural and intuitive technologies and imagining how it would be like if they became reality. Now all of those fantasies start coming to life.

We also wanted to step into the future and try to create something new and innovative. A goal was to create a simple yet functional application, which can be controlled using hand motion as the main input. This application would also have to serve a practical yet entertaining purpose. Combining this goals lead us to the idea of the Japanese kana learning tool.

In the first chapter we give an overview of the technologies and techniques used in this thesis. We explain methods and capabilities of two main technologies - Leap Motion and Pygame, which are later used in the creation of the application. We also give the reader a quick overview of the Japanese *hiragana*.

Second chapter explains in detail the work of our stroke detection method. The chapter describes feature point extraction, which is crucial to the method of stroke detection. After that we explain how the drawn stroke comparison with the original stroke pattern works.

Third chapter gives an overview of the written application. It explains how the original stroke patterns are stored and used. It also describes the created control system for Leap Motion and keyboard and shows how the drawing and visuals work.

In the last chapter we describe a few tests we did with the application and their results.

2 Overview

In this section we wanted to give the reader a basic understanding and knowledge of the technologies and techniques used in this thesis.

Two main technologies we used to write this work were Pygame 1.9.1 and Leap Motion SDK Windows 2.2.3. In next sections we will give a basic overview on these technologies.

As the main programming language Python 2.7 was chosen due to its easy and readable syntax and the compatibility with Pygame and Leap Motion.

Please note that all of the used software, libraries and plugins work with given versions and are not guaranteed to work under different conditions, such as different versions or platforms.

2.1 Leap Motion

Leap Motion is a small device, only 13 x 13 x 76 mm and 45 grams [1], equipped with optical sensors and infrared light that allow the system to recognize and track hands, fingers and finger-like tools [2].

Figure 1. Leap Motion [1]

The Leap Motion Controller tracks all 10 fingers up to 1/100th of a millimetre at a rate of over 200 frames per second. It's dramatically more sensitive than existing motion control technology [1].

The sensors are directed along the y-axis – upward when the controller is in its standard operating position – and have a field of view of about 150 degrees. The effective range of the Leap Motion Controller extends from approximately 25 to 600 mm above the device (1 inch to 2 feet) [2]. The only restriction is that Leap Motion has the fixed orientation, so it has to be has to be facing upwards with x-axis towards the user. That means, that if it is placed in different orientation, for example upside down or sideways, then the Leap Motion software cannot detect or adjust to that.

Figure 2. The Leap Motion controller's view of your hands [2]

Leap Motion can be run on three main operation systems: Windows, Mac, Linux. And to run Leap Motion, it only needs, in addition to one of those operating systems, Leap Motion software and the USB port.

Leap Motion supports many different programming languages like C#, C++, Java, Python, Objective-C and JavaScript. In addition it has plugins to work with game engines Unity 3D and Unreal Engine 4. It also has some other platform integrations and libraries.

2.1.1 Motion tracking

Leap Motion controller tracks hands, fingers and tools in its field of view and presents tracked information to the application as a `Frame` object. Each `Frame` object representing a frame contains lists of tracked entities, such as hands, fingers, and tools, as well as recognized gestures and factors describing the overall motion in the scene. The `Frame` object is essentially the root of the Leap Motion data model.

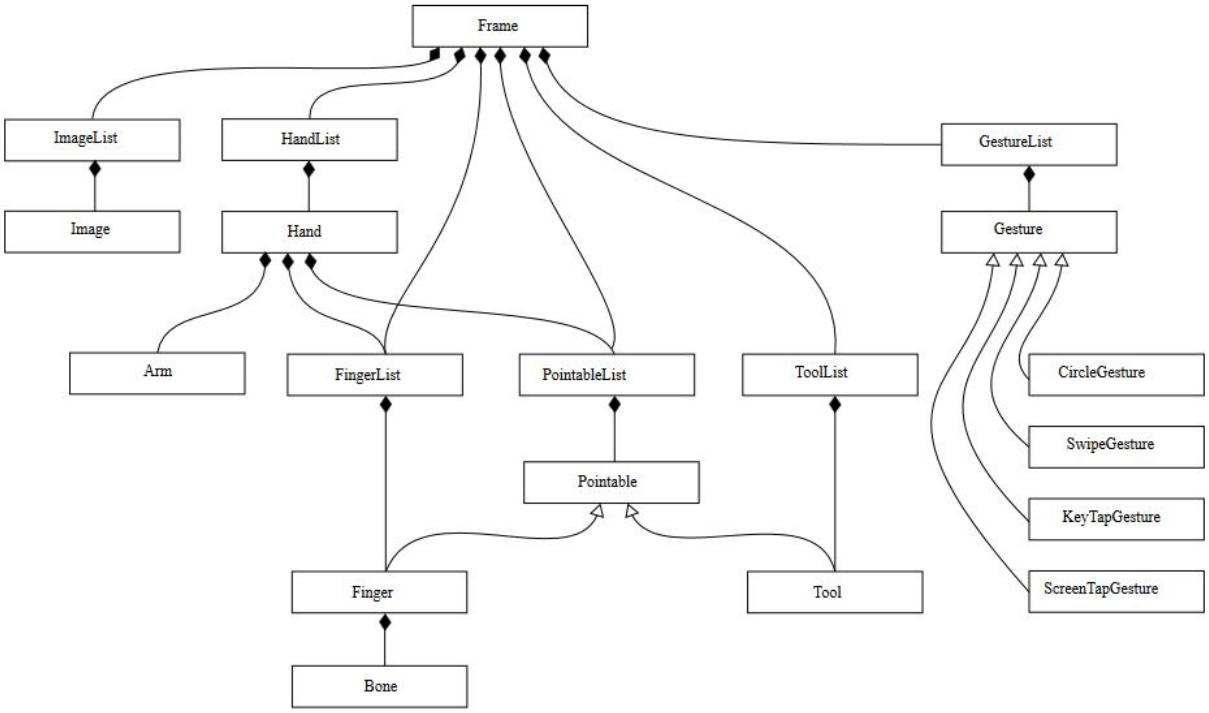

Figure 3. Tracking model [3]

Hands are the main entity tracked by the Leap Motion controller and is represented by the `Hand` class, which provides access to hand position and other information about the hand as well as the arm to which the hand is attached, and list of all the fingers. There could be more than two `Hand` objects in one `Frame`, but it is not recommended for tracking quality.

The Leap Motion software combines its sensor data with an internal model of the human hand to help cope with challenging tracking conditions, for example when some of the fingers are hidden from Leap Motions view. Detection and tracking work best when the controller has a clear, high-contrast view of an object's silhouette [2]. This, however, means that if the hand is

badly visible then the software can make mistakes and return the wrong information, for example identifies right hand as the left one.

The Leap Motion controller provides information about each finger on a hand. If all or part of a finger is not visible, the finger characteristics are estimated based on recent observations and the anatomical model of the hand. Fingers are identified by type name, i.e. *thumb*, *index*, *middle*, *ring*, and *pinky* [2].

Fingers are represented by the `Finger` class, which is a kind of `Pointable` object and provides information on fingers, like position, direction and even all the bones from the finger with their positions and directions.

Leap Motion also tracks tools, which can only be thin, cylindrical objects as pencil for example. Tools are represented by `Tool` class, which is also kind of `Pointable` object and provides its position and direction.

2.1.2 Gestures

The Leap Motion software recognizes certain movement patterns as gestures which could indicate a user intent or command. Gestures are observed for each finger or tool individually. The Leap Motion software reports gestures observed in a frame the in the same way that it reports other motion tracking data like fingers and hands [4].

Gestures are represented by the `Gesture` class and if occurred are returned by the `Frame` object just like `Hand` and `Finger` objects are. `Gesture` class has subclasses for all four gestures it can recognize: `CircleGesture`, `KeyTapGesture`, `ScreenTapGesture` and `SwipeGesture`. All of the subclasses contain information related to the specific gesture.

The following movement patterns are recognized by the Leap Motion software:

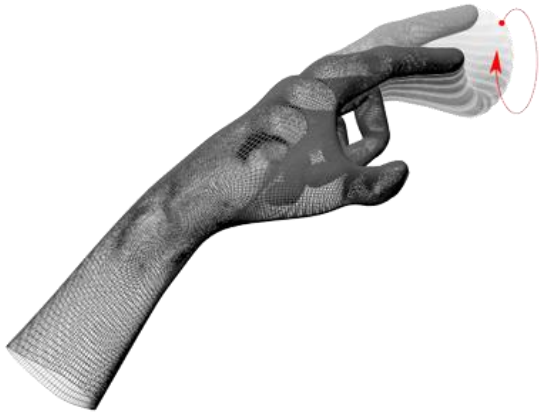

Figure 4. Circle — A finger tracing a circle [4]

Figure 6. Key Tap — A tapping movement by a finger as if tapping a keyboard key [4]

Figure 5. Swipe — A long, linear movement of a hand and its fingers [4]

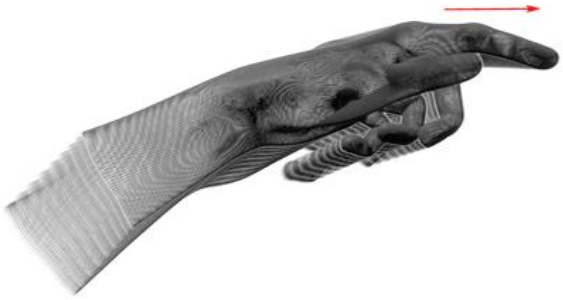

Figure 7. Screen Tap — A tapping movement by the finger as if tapping a vertical computer screen [4]

The gestures Circle and Swipe are continuous. The Leap Motion software updates the progress of these gestures each frame. Taps are discrete gestures. The Leap Motion software reports each tap with a single `Gesture` object [4].

2.1.3 Coordinate system

The Leap Motion Controller provides coordinates in units of real world millimetres within the Leap Motion frame of reference. That is, if a finger tip's position is given as `(x, y, z) = [100, 100, -100]`, those numbers are millimetres – or, x = +10cm, y = 10cm, z = -10cm [5].

The Leap Controller hardware itself is the centre of this frame of reference. The origin is located at the top centre of the hardware (as shown in Figure 8). That is if you touch the middle of the Leap Motion controller (and were able to get data) the coordinates of your finger tip would be [0, 0, 0] [5].

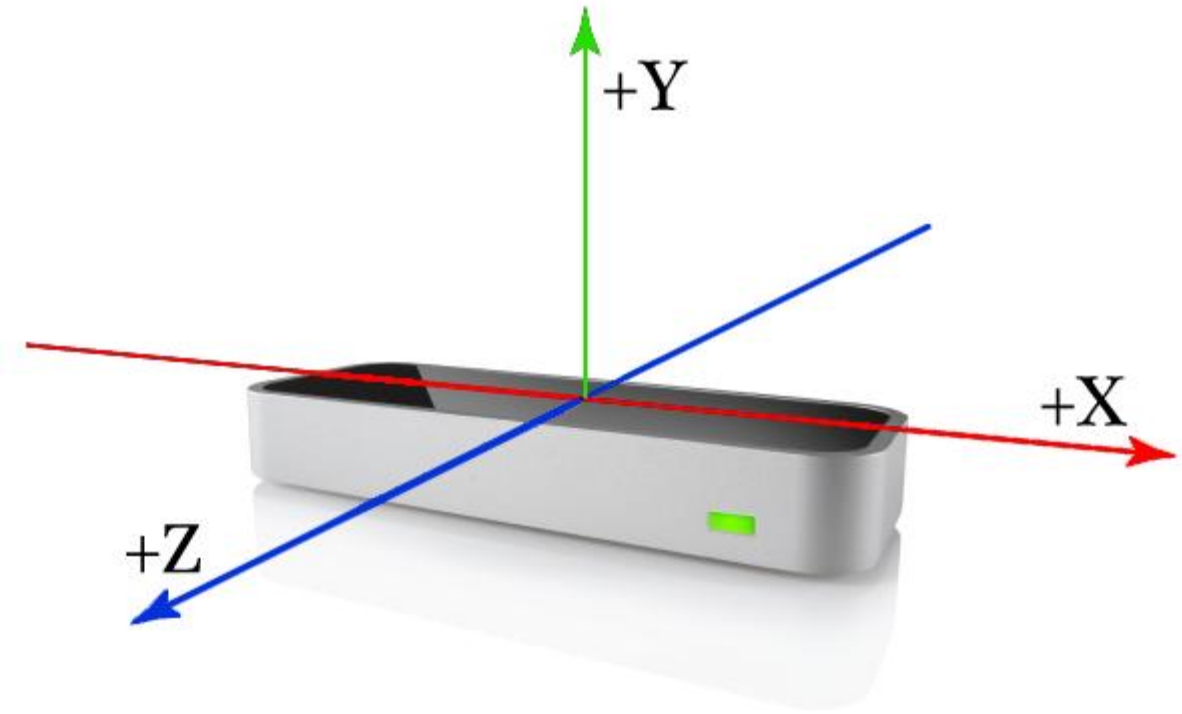

Figure 8. The Leap Motion right-handed coordinate system [5]

Due to the fact, that Leap Motion Controller provides coordinates in millimetres they have to be interpreted so that they make sense in the application. Differences between coordinate systems may require flipping axes and neglecting data from some of them. For example, most 2D applications put origin at the top, left corner of the window, with y values increasing downward. But Leap Motion origin point is at the top center of the Leap Motion itself, with y values increasing upwards.

Leap Motion has pretty large field of view, but it is shaped as an inverted pyramid, so the available range on the x and z axes is much smaller close to the device, than it is near the top of the range. That means if the whole field of view is used, then user might not be able to reach some parts of the application.

Most of those problems can be solved by using built in coordinate normalizer `InteractionBox`. `InteractionBox` converts coordinates from Leap Motion coordinates to normalized scale running between 0.0 and 1.0 and can be then easily converted to application appropriate scale. It also defines a rectangular area within the Leap Motion field of view so as long as the user's hand or finger stays within this box, it is guaranteed to remain in the Leap Motion field of view and be suitable for the application.

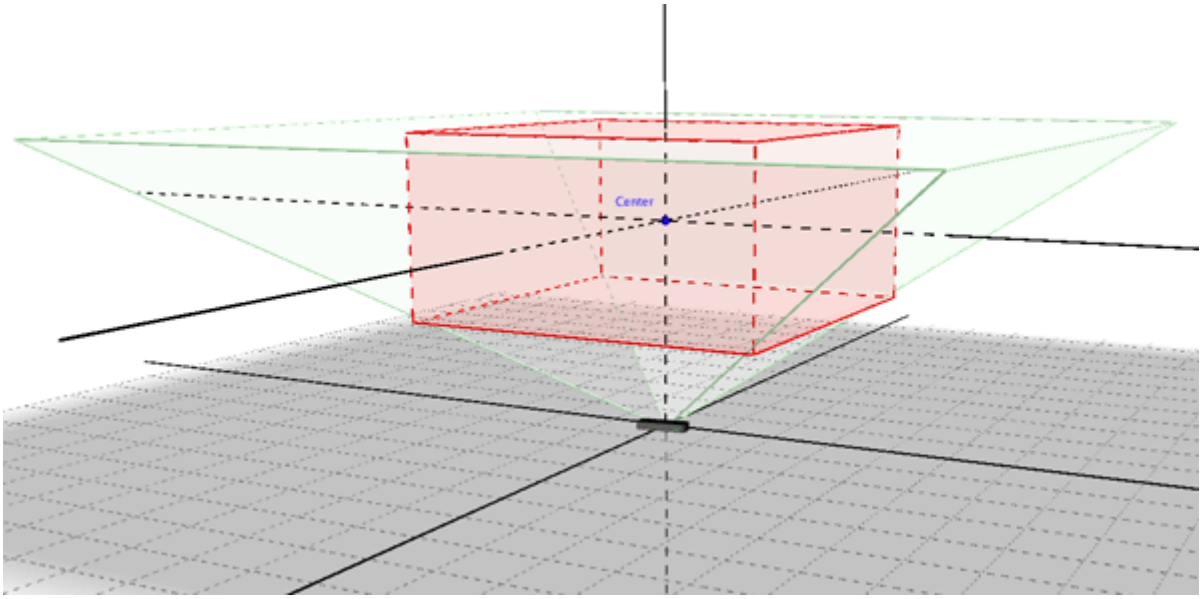

Figure 9. InteractionBox field [5]

The size of the `InteractionBox` is determined by the Leap Motion field of view and the user's interaction height setting (in the Leap Motion control panel). The controller software adjusts the size of the box based on the height to keep the bottom corners within the field of view. If user sets the interaction height higher, then the box becomes larger. A user can also set the interaction height to adjust automatically. If the user moves his or her hands out the interaction box, the controller software readjusts the box height [5].

Every frame provides new interaction box. Because the interaction box can change over time, the normalized coordinates of a point measured in one frame may not match the normalized

coordinates of the same real world point, when normalized using the interaction box provided by another frame. Thus straight lines or perfect circles “drawn in the air” by the user may not be straight or smooth if they are normalized across frames [5].

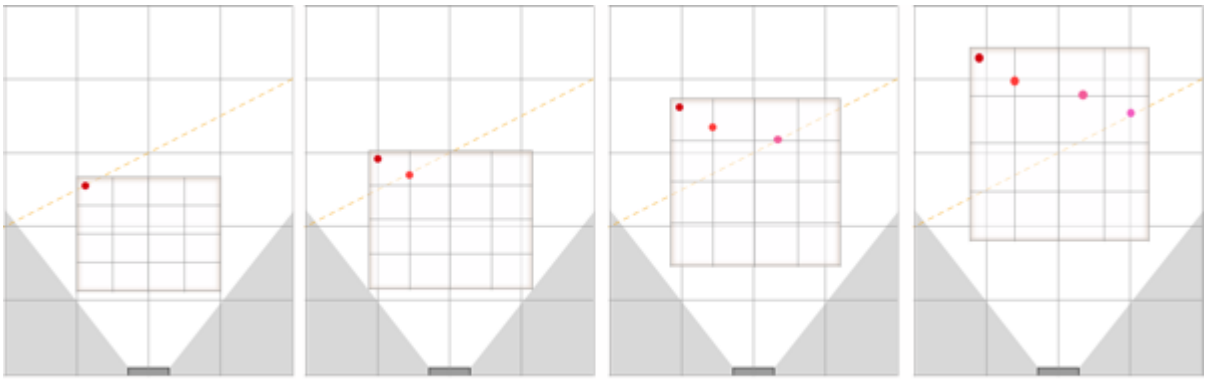

Figure 10. Change in interaction box [5]

To guarantee that a set of points are normalized consistently, the same `InteractionBox` object has to be used for normalizing.

2.2 Pygame

Pygame is a set of free Python modules designed for writing games. Released under the GPL License, you can create open source, free, freeware, shareware, and commercial games with it [6].

Pygame adds functionality on top of the excellent SDL library [6]. Simple DirectMedia Layer (SDL) is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. SDL 2.0 is distributed under the zlib license. This license allows you to use SDL freely in any software [7].

To avoid common OpenGL setup issues, Pygame does not require OpenGL, it uses either OpenGL, DirectX, WinDIB, X11, Linux Frame Buffer, and many other different backends, including an ASCII art backend [6].

2.2.1 Main features

Pygame uses optimized C, that is often 10-20 times faster than python code, and Assembly code, that can easily be 100x or more times faster than python code, for core functions [6].

Pygame supports use of multi core CPUs. Selected Pygame functions release the dreaded python GIL (global interpreter lock), which is something that can be done from C code [6].

Pygame is highly portable supporting vast variety of operating systems like Linux, Windows, Mac, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX. Code olso contains support for AmigaOS, Dreamcast, Atari, AIX, OSF/Tru64, RISC OS, SymbianOS, and OS/2, but these are not officially supported. It can also be used on handheld devices from Nokia, game consoles like gp2x, and the One Laptop Per Child (OLPC) [6].

2.2.2 Modules

The `pygame` package represents the top-level package for others to use. Pygame itself is broken into many submodules, but this does not affect programs that use Pygame [8].

A `pygame.Surface()` object is used to represent any image. The *Surface* has a fixed resolution and pixel format. *Surfaces* with 8bit pixels use a colour palette to map to 24bit colour [9].

Control over Pygame display is managed by `pygame.display` module. *Display* is created as any other `pygame.Surface()` object, with all of its functions, but there can be only one display at a time. If another display is created, Pygame closes the first one. The origin of the display, where x = 0, and y = 0 is the top left of the screen. Both axis increase positively towards the bottom right of the screen [10].

The `pygame.font` module allows for rendering TrueType fonts (.ttf) into a new *Surface* object. The actual font rendering is done by the `pygame.font.Font()` object. The render

can emulate bold or italic features, but it is better to load from a font with actual italic or bold glyphs. The rendered text can be regular strings or Unicode [11].

Pygame also enables user do draw several simple shapes, like rectangles, polygons, cyrcles, ellipses, arcs and lines, to *Surface* objects. For that it has `pygame.draw` module [12].

Pygame handles all events through `pygame.event` module. The input queue is heavily dependent on the `pygame.display` module. If the *display* has not been initialized and a video mode not set, the event queue will not really work.

All events have a type identifier. An `pygame.event.EventType()` event object contains an event type identifier and a set of member data. *EventType* objects are retrieved from the pygame event queue. You can create your own new events with the `pygame.event.Event()` function [13].

`pygame.key` module contains functions for dealing with the keyboard. Event queue gets `pygame.KEYDOWN` and `pygame.KEYUP` events when the keyboard buttons are pressed and released. Both events have a key attribute that is an integer id representing every key on the keyboard [14].

The `pygame.mouse` functions can be used to get the current state of the mouse device. When the display mode is set, the event queue will start receiving mouse events. The mouse buttons generate `pygame.MOUSEBUTTONDOWN` and `pygame.MOUSEBUTTONUP` events when they are pressed and released. These events contain a button attribute representing which button was pressed [15].

For framerate management there is a `pygame.time` module. Time in Pygame is represented in milliseconds (1/1000 seconds) [16].

2.3 Hiragana

Hiragana is a Japanese syllabary, one basic component of the Japanese writing system, along with katakana, kanji, and in some cases rōmaji (the Latin-script alphabet). The word *hiragana* means "ordinary syllabic script" [17].

Hiragana and katakana are both kana systems. With one or two minor exceptions, each sound in the Japanese language is represented by one character in each system. This may be either a vowel such as *"a"* (*hiragana* あ); a consonant followed by a vowel such as *"ka"* (か); or *"n"* (ん), a nasal sonorant which, depending on the context, sounds either like English *m*, *n*, or *ng* ([ŋ]), or like the nasal vowels of French. Because the characters of the kana do not represent single consonants (except in the case of ん "n"), the kana are referred to as syllabaries and not alphabets [17].

2.3.1 Writing system

The modern *hiragana* syllabary consists of 46 characters:

- 5 singular vowels
- Notionally, 45 consonant–vowel unions, consisting of 9 consonants in combination with each of the 5 vowels, of which:
- 1 singular consonant

These are conceived as a 5×10, as illustrated in the Figure 11, with the extra character being the anomalous singular consonant ん (*N*) [17].

あ	い	う	え	お
a	i	u	e	o
か	き	く	け	こ
ka	ki	ku	ke	ko
さ	し	す	せ	そ
sa	shi	su	se	so
た	ち	つ	て	と
ta	chi	tsu	te	to
な	に	ぬ	ね	の
na	ni	nu	ne	no
は	ひ	ふ	へ	ほ
ha	hi	fu	he	ho
ま	み	む	め	も
ma	mi	mu	me	mo
や		ゆ		よ
ya		yu		yo
ら	り	る	れ	ろ
ra	ri	ru	re	ro
わ	を	ん		
wa	wo	n/m		

Figure 11. *Hiragana* table

3 Stroke detection

Stroke detection is a method that compares drawn strokes and original stroke patterns with each other and determines their compatibility.

A stroke is one continuous line, that consists of one or more linked line segments. It is defined by a sequence of points. Point is one single dot with x and y coordinates. Points make up strokes and line segments. Part of the stroke, that is made up by two points, one at the start and one at the end of it, is called line segment. They have important role in stroke comparison.

Original stroke pattern is number of points that make up the right pattern of strokes. These patterns of strokes are used as an reference standard and are considered the correct shape of the characters. The drawn strokes are thereafter compared against these original strokes.

Character or sign is a sequence of strokes, which make up one complete entity.

3.1 Extracting feature points

Leap Motion can process over 200 frames per second and the application runs, at least, at 60 frames per second. Considering, that the application takes one frame from Leap Motion every cycle, this calculates to a whole lot of coordinate points.

In order to effectively compare data points from the Leap motion with the original stroke pattern, there is a need to extract the essential coordinate points which make up the stroke.

3.1.1 Basis of point extraction

Currently there are several methods and techniques employed for optimizing coordinate lists. One such technique has been described below.

Feature points are extracted by such a method as Ramner. First, the start and end points of every stroke are picked up as feature points. Then, the most distant point from the straight line between adjacent feature points is selected as a feature point if the distance to the straight line

is greater than a threshold value. This selection is done recursively until no more feature points are selected. The feature point extracting process is shown in Figure 12 [18].

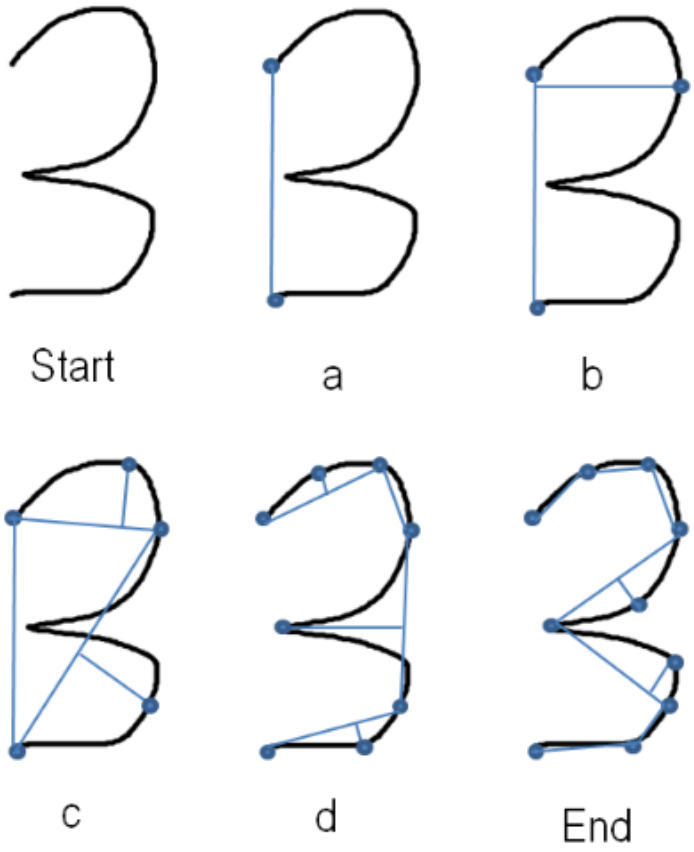

Figure 12. Feature points extraction [18]

Though this method is very good for extracting feature point for character recognition, it does not suite the needs of this thesis. The amount of coordinate points has to be predetermined by the original stroke pattern and this method fails to provide a suitable solution in such a case.

3.1.2 Improved method

To compare the strokes easily and efficiently we need to extract the same amount of points that is given in the original stroke pattern.

First, the start and the end point are chosen for the first two feature points. Then, the furthest point from the straight line, which can be drawn between those two points, is chosen as the next feature point and added to the list of feature points. Next, furthest points from the lines,

which can be drawn between the closest feature points, are picked and the one that had the longest distance from the line is chosen as the feature point. This runs recursively until the needed amount of points is reached.

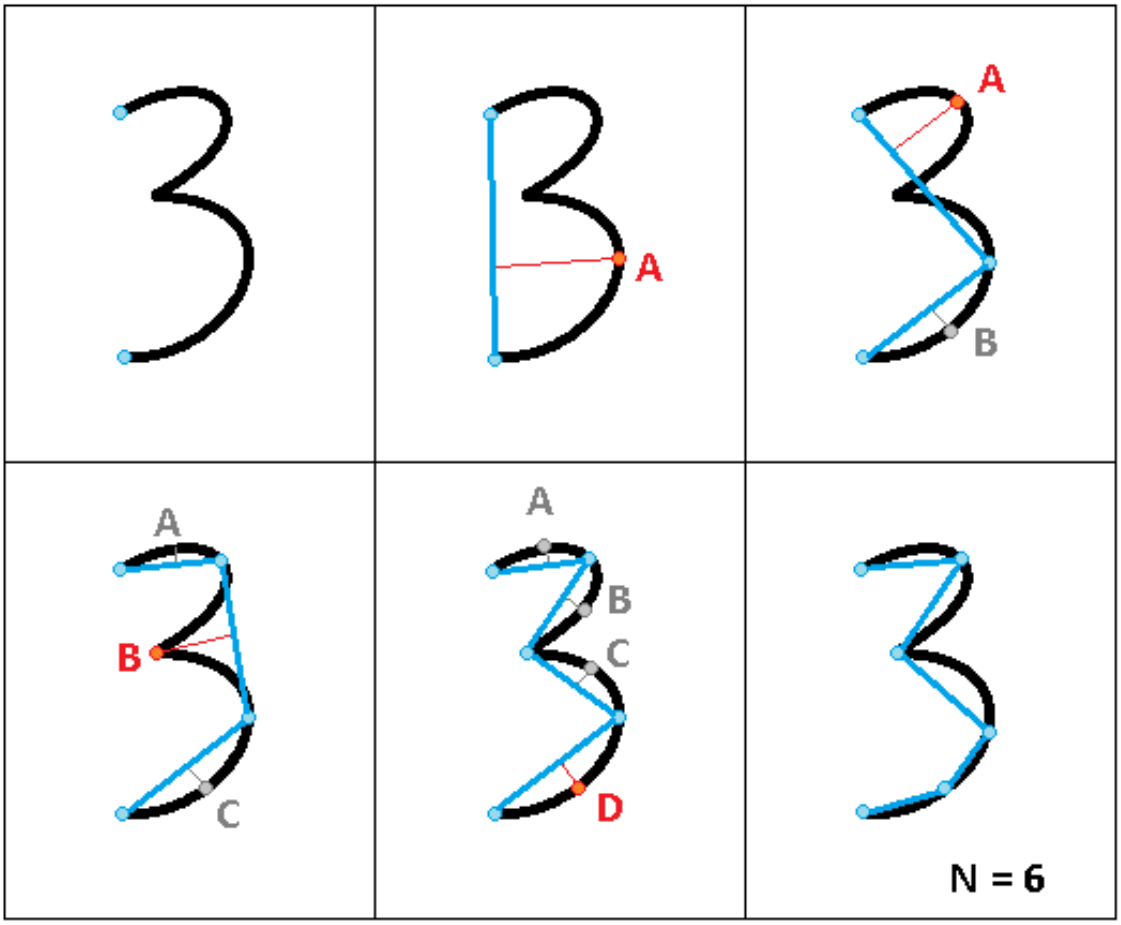

Figure 13. Improved method for point extraction

Note that this algorithm runs only till the needed amount of points is chosen. So if only two points are needed, the algorithm stops after getting the start and the end point.

Next the code for the method is shown:

```
cord = []
cord.append(line[0])
cord.append(line[-1])

pos = []
pos.append(0)
pos.append(len(line)-1)

while len(cord) < nr:
     max_dist = 0
     max_point = (0, 0)
     max_i = 0
     max_pos = 0
     for i in range(len(cord)-1):
          for j in range(pos[i], pos[i+1]):
               dist = get_distance_to_line(cord[i], cord[i+1],
line[j])
               if max_dist < dist:
                    max_dist = dist
                    max_point = line[j]
                    max_i = i + 1
                    max_pos = j
     cord.insert(max_i, max_point)
     pos.insert(max_i, max_pos)

return cord
```

Finding the distance

To find the distance between the line and the point we have to find the point on the line, which is closest to the given point, and get the distance between those two.

Let's say that line is created between point A and B. The point we want to get the distance to, is P. In that case, we get vectors from A to B and from A to P. Then we get u, the scaling factor of the projection of AP onto AB, which will help us get the coordinates of the closest point on the line AB to point P [19]. If the projection was not made on the line segment AB (u < 0 or u > 1), then we set it to either end point of the line segment. Then we get the coordinates of the closest point on the line to the point P by offsetting the A coordinates and find the distance between those and point P.

$$A(x_1, y_1) \qquad B(x_2, y_2) \qquad P(x_p, y_p)$$

$$\overrightarrow{AB}(x_{AB}, y_{AB}) \qquad \overrightarrow{AP}(x_{AP}, y_{AP})$$
$$x_{AB} = x_2 - x_1 \qquad x_{AP} = x_p - x_1$$
$$y_{AB} = y_2 - y_1 \qquad y_{AP} = y_p - y_1$$

$$u = \frac{x_{AP} * x_{AB} + y_{AP} * y_{AB}}{{x_{AB}}^2 + {y_{AB}}^2}$$

$$u > 1 \rightarrow u = 1$$
$$u < 0 \rightarrow u = 0$$

$$x = x_1 + u * x_{AB}$$
$$y = y_1 + u * y_{AB}$$

$$dist = \sqrt{(x - x_p)^2 + (y - y_p)^2}$$

3.2 Comparing lines

Given line comparison works only in case if both lines have the same amount of line segments. For that we use the previous method of extracting feature points.

We wanted to make this method independent from the size and the position in the grid of the new line compared to the original stroke pattern. So for that we compare stroke with the original stroke pattern by their angles and line segment lengths.

Comparing angles is easy since they are independent from line scale and position. With line segments it is more difficult, so we use *distance difference*. *Distance difference* is difference between drawn line segment length and the length of the same segment from original stroke pattern. That way we can compare if the line segments are all in the same relation to each other as they are in the original stroke pattern. For that *average distance difference* is used. It signifies how big is the line segments average *distance difference*.

Considering that the new sign cannot be identical to the original stroke pattern, we use *allowed error*. Allowed error is a constant that signifies how much error we allow in our comparisons. Its value is given from 0 to 1.

```
if (len(line1) == len(line2) and len(line1) != 0):
     n = len(line1) - 1
     dist_dif_sum = 0

     for i in range(n):
          angle1 = get_angle_with_x_axis(line1[i], line1[i+1])
          angle2 = get_angle_with_x_axis(line2[i], line2[i+1])
          if are_similar_angles(angle1, angle2, allowed_error)
== False:
               return 0

          dist1 = get_distance(line1[i], line1[i+1])
```

```
            dist2 = get_distance(line2[i], line2[i+1])
            dist_dif_sum += get_distance_dif(dist1, dist2)
            average_dist_dif = dist_dif_sum / (i+1)
            if are_similar_distances(dist1, dist2,
average_dist_dif, allowed_error) == False:
                                    return 0


    average_dist_dif = dist_dif_sum / len(line1)
```

3.2.1 Angle comparison

Angle for comparison is found between vector from A to B and the positive x-axis. The result is in radians between `-pi` and `pi`.

$$A(x_1, y_1) \qquad B(x_2, y_2)$$
$$\overrightarrow{AB} = (x_{AB}, y_{AB})$$
$$x_{AB} = x_2 - x_1$$
$$y_{AB} = y_2 - y_1$$
$$angle = \arctan(\frac{y_{AB}}{x_{AB}})$$

Angles are compared based on *allowed error*. If difference in angles remains in the range of *allowed error,* then the angles are considered similar.

3.2.2 Distance comparison

Euclidean metric is used to get distance between two points

$$A(x_1, y_1) \qquad B(x_2, y_2)$$

$$\overrightarrow{AB} = (x_{AB}, y_{AB})$$

$$x_{AB} = x_2 - x_1$$

$$y_{AB} = y_2 - y_1$$

$$dist = \sqrt{x_{AB}{}^2 + y_{AB}{}^2}$$

Line segments are compared considering the *allowed error* and the *average distance difference*. They are considered similar if the difference between *distance difference* and *average distance difference* is in range of *allowed error*.

3.3 Comparing characters

The different lines or strokes of the characters should be drawn in a certain order. When comparing the stroke lines with original stroke pattern that order is considered. So if the drawn line is not deemed similar to the same ordered line in the original stroke pattern, it is returned as the wrong line.

Also the direction of the stroke is important. If the drawn stroke is in the different direction from the one in the original stroke pattern then it is also considered wrong.

Character comparison also tries to determine if all the strokes are similar in size compared to each other.

4 Application for stroke detection

The stroke detection method was created in order to utilize it in a specific Leap Motion application. This application is for leaning to write Japanese kana, but can also be used to learn to write any character which uses strokes.

The main parts of the applications are:

- Usage of original stroke patterns
- Application control methods
- Drawing and visuals

4.1 Original stroke patterns

Original stroke patterns have to be stored in a certain way in character sheet for the application to be able to read them. Character sheet is the whole list of original stroke patterns that are read into the application at one time.

After the character sheet is read into the application, all characters have to be set to their right positions and the character widget has to be created, so that the user can choose the character for learning.

4.1.1 Stroke pattern storage structure

We wanted to create universal application that could potentially be used with any original stroke patterns, not only Japanese *hiragana*. For that purpose we decided to use external file for storing the original stroke patterns (referred also as character sheet). For its readability and easy parsing XML file was chosen.

The particular file [20] used in this application was released under the GNU Lesser General Public License version 2.1 [21].

For application to be able to parse given XML file correctly, it needs to have a particular structure. Name of the root element of the document is not important in this application. Characters have to be stored as children of the document root element and be stored in <character> elements.

<character> element needs to have two children: <utf8> and <strokes>. First is to store the character in UTF-8 coding and the second one for containing the strokes for original stroke parent of that particular character. In case of setting a different XML file, that does not contain character stroke patterns, then the description of the element in UTF-8 coding can be set as the <utf8> element contents.

Element <strokes> can have infinite amount of children elements <stroke>. Every <stroke> element represents single stroke pattern of the character. Its children <point> represent points and have must have attributes “x” and “y” as integer valued coordinates.

Here is an example of one character stroke pattern in XML:

```
<character>
     <utf8>&#x304D;</utf8>
     <strokes>
          <stroke>
               <point x="256" y="260"/>
               <point x="683" y="266"/>
          </stroke>
          <stroke>
               <point x="300" y="500"/>
               <point x="736" y="526"/>
          </stroke>
          <stroke>
               <point x="430" y="90"/>
               <point x="513" y="623"/>
          </stroke>
```

```
            <stroke>
                <point x="270" y="683"/>
                <point x="346" y="823"/>
                <point x="533" y="890"/>
            </stroke>
        </strokes>
</character>
```

4.1.2 Character position

The original stroke pattern might have the character (referred also as sign) position anywhere on the screen. For better usage of it, we wanted to show it in the middle of the application window. Because it is not drawn on the different *Surface* object, but on the screen in series of lines we needed to alter the coordinates of each stroke.

For that, after creating the original character, we find the top left and bottom right points of the minimal rectangle the character could fit in. Next, we find centre of the application window and subtract half of the width and height of the rectangle to get the right position for the character rectangle to start from. By dividing coordinates of the right character starting point and the current one we get the difference in coordinates. Adding that to the original sign coordinates we get the right position for the sign.

4.1.3 Original stroke pattern selection

We wanted user to be able to select the character they want to learn to draw, so for that purpose we created character choosing widget. Considering, that we decided to make original stroke patterns easily changeable, we had to make choosing widget also adaptable.

The idea was to create a button panel that is controllable with Leap Motion.

For button outlines to be clear and visible, we wanted to create a border around every character button. Pygame does not have the functionality to create borders around *Surface*

objects, so to create them we have to draw two different *Surface* objects. We first draw the bigger one and set its colour the same we want our borders to be. Next, we draw different surface, smaller by the size of the wanted border and the colour we want our character selection button to be. After that we draw smaller *Surface* of top of the border coloured one.

In Pygame, writing directly onto a *Surface* object is not possible, so to write character onto created black button we need to create text *Surface* and draw it onto the black button *Surface*. Note here, that Pygame does not have fonts suitable for rendering Japanese characters, so we had to import a new font Japanese character friendly font. In case of changing character sheet, font might not be able to render that and buttons will not have readable text on them.

To create the character button widget, first the buttons for all the characters from character sheet are created. Then they are drawn on a *Surface* object in rows or columns according to the program's specifications. Example is shown in Figure 14. Character widget is set as the main part of the home screen.

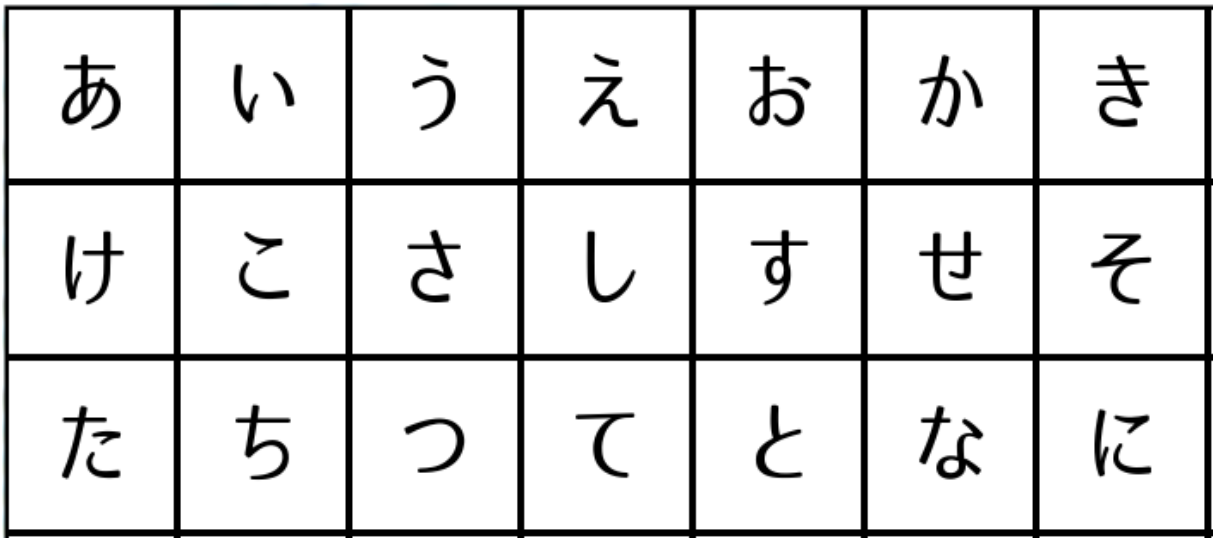

Figure 14. Character widget

For creating a character widget, application uses information from character sheet. So if the character sheet is changed, the content of the character widget will be different as well.

To detect the pushed button we ask all buttons if the pushed coordinates were theirs, until we find the right one and return the button. Controller then finds the appropriate sign and sets it as the sign for learning and sends user to drawing view, where the sign is drawn in the middle

of the screen for user to see and try to copy. Character is drawn in light grey colour, so that it does not get mistaken it with the one user is currently drawing.

4.2 Leap Motion controls

Leap Motion controls are used as main control method in this application. They allow user to move the application cursor, push and draw onto the screen and use different gestures for different actions.

Leap Motion enables to track not only fingers but also pen-like tools. They are all classifies as *Pointable* objects and reports its physical characteristic, like position, type, touch zone etc. In this application usage of both is supported.

For tracking, the closest object to the screen is chosen with the help of the Leap Motion method `frame.pointables.frontmost`. If there are many *Pointables* close to the screen, then the Leap Motion controller can make mistakes in interpreting the right *Pointable* object and the cursor might jump unexpectedly.

4.2.1 Converting coordinates

Leap Motion returns the coordinates in millimetres so they have to be normalized. Leap Motion provides a method for that. `InteractionBox` converts coordinates from Leap Motion coordinates to normalized scale running between 0.0 and 1.0 and can be then converted to application appropriate scale.

Every Leap Motion frame provides new `InteractionBox` object, but due to the fact that it can adjust and move, coordinates normalized with different `InteractionBox` every turn can be different and the point can start to flicker. To prevent that, we save a single `InteractionBox` at the start of the application and keep using it during the whole work cycle. If the user wants to make adjustments to the interaction box, then the application has to be closed and restarted.

After the `InteractionBox` has normalized the coordinates, it returns them in range from 0 to 1, so we still had to convert them to a readable form for the application. For that we multiply the normalized the x coordinate with application width and the y coordinate with the height of the application.

Y–axis in `InteractionBox` starts from bottom and increases towards the top on the view range of the Leap Motion. In the application, on the other hand, the y-axis origin point is on top and increases towards the bottom of the application. So for them to match we had to flip the y-axis. For that we have to subtract the normalized y coordinate from 1 before we multiply it by the application height.

```
self._pointable = self._frame.pointables.frontmost
if self._pointable.is_valid:
     leap_point = self._pointable.stabilized_tip_position
     normalized_point = self._iBox.normalize_point(leap_point,
True)


app_x = int(round(normalized_point.x * APP_WIDTH))
app_y = int(round((1 - normalized_point.y) * APP_HEIGHT))
```

4.2.2 Touch emulation

The application has only 2D controls, but we still used z-axis for touch emulation. For that Leap Motion software has two methods in `Pointable` class: `touch_zone()` and `touch_distance`.

The first method is for determining if the *Pointable* has passed through the floating touch plane that adapts to user's finger or tool. If the *Pointable* is past the plane, then it is in „touching" zone, otherwise it is in the „hovering" zone. If the *Pointable* is not visible, then the zone is „none". We used that functionality to determine if the user is touching the screen or not.

The second method returns the touch distance in range from -1 to 1. The value 1.0 indicates the *Pointable* is at the far edge of the hovering zone. The value 0 indicates the *Pointable* is just entering the touching zone. A value of -1.0 indicates the *Pointable* is firmly within the touching zone. Values in between are proportional to the distance from the plane. Thus, the `touch_distance` of 0.5 indicates that the *Pointable* is halfway into the hovering zone [22].

This last method enabled us to create the functionality for warning user about getting close to the transfer zone. When the user is in „hovering“ zone, then before reaching the „touching“ zone, the cursor becomes dark grey, warning the user, that it will soon transfer to the „touching“ zone and the application will register the „click“. Also in reverse, when the user is in the „touching“ zone, cursor will become dark grey prior to leaving the zone.

4.2.3 Gestures

Leap Motion offers a range of built-in hand gestures in order to allow for easy control over the unit and applications.

By default Leap Motion gestures are disabled. In this application we enabled three of the Leap Motion gestures: *Circle*, *Swipe* and *Key tap*.

Commands Leap Motion gestures trigger:

- *Circle* – delete the last drawn line
- *Swipe* – return to the home screen
- *Key tap* – compare characters

While drawing user might make some movements, which could be interpreted by Leap Motion controller as gestures. To prevent these unwanted interruptions, we enabled the handling of the Leap Motion gestures only while user is not writing anything and is just hovering over the screen.

Application also makes sure that every gesture is carried out only once. For that it saves the ID of the gestures from the last Leap Motion frame and compares them with the ID’s of the new gestures. In addition it prevents continuous gestures like *Circle* and *Swipe* to trigger more

than one action per time. It means that gestures that are added to every frame as long as they last trigger the action still only once. To trigger the action again, user has to stop the movement and start a new one. For example, that prevents the user from deleting lines too fast and losing more lines than the user intended.

Before triggering the comparison of the characters, application checks if the needed amount of lines is drawn to the screen. We decided not to make the comparison start automatic, in case user wants to redraw the last line.

4.3 Mouse and keyboard controls

For testing purpose it is also possible to use the application with a standard mouse. Before getting coordinates from Leap Motion, application checks if the Leap Motion is connected to the system. In case there is no connection to Leap Motion, application switches to the mouse controlled mode.

Additionally, gesture activated actions can also be activated using a standard keyboard. This functionality works independently from the connection to Leap Motion.

4.3.1 Mouse controls

In case the Leap Motion is not connected to the system, user can use the application with the help of the mouse.

For interacting with mouse we used `pygame.mouse` module. With it we were able to obtain coordinates of the mouse and check every cycle if the mouse button is pressed. With that we were able to imitate the Leap Motion controls using the mouse.

4.3.2 Keyboard events

We also added keyboard events as an alternative control method to the Leap Motion gestures. Performing Leap Motion gestures might sometimes be tricky, but with the keyboard alternative is user still able to perform all needed tasks.

Keyboard commands trigger:

- Delete – deletes whole drawn character
- Backspace – deletes last drawn line
- Space – compare characters
- Home – return to home screen

Keyboard commands are realized through `pygame.KEYDOWN` event that are added to the event queue when the keyboard buttons are pressed. Those events have key attribute that represents a key on the keyboard. If the right key is hit, the certain events are triggered.

```
for event in pygame.event.get():
     if event.type == pygame.KEYDOWN:
          if event.key == pygame.K_SPACE:
               self.compare_signs()
          if event.key == pygame.K_DELETE:
               self.clear_sign()
          if event.key == pygame.K_HOME:
               self.back_to_home_screen()
          if event.key == pygame.K_BACKSPACE:
               self.drawn_sign.delete_last_line()
```

4.4 Character drawing

While the character is being drawn by the user, the application normalizes every stroke after it is drawn. When the drawing is complete and the user submits the result, the application compares the drawn character to the original stroke pattern and gives feedback.

4.4.1 Drawn stroke normalization

Controlling the application with Leap Motion can be a bit of a challenge for many reasons. First, it is not the easiest task to draw nice straight lines in the air. Secondly, Leap Motion is not perfect, so it can and does make mistakes. For that reason, after the stroke is drawn, the application normalizes it, to try and make our stroke look as much alike with the expected one as possible.

After the line is drawn by the user, obtained coordinates are run through the feature point extraction method considering the needed number of points for expected stroke. Old coordinates of the drawn stroke are replaced by the extracted feature points.

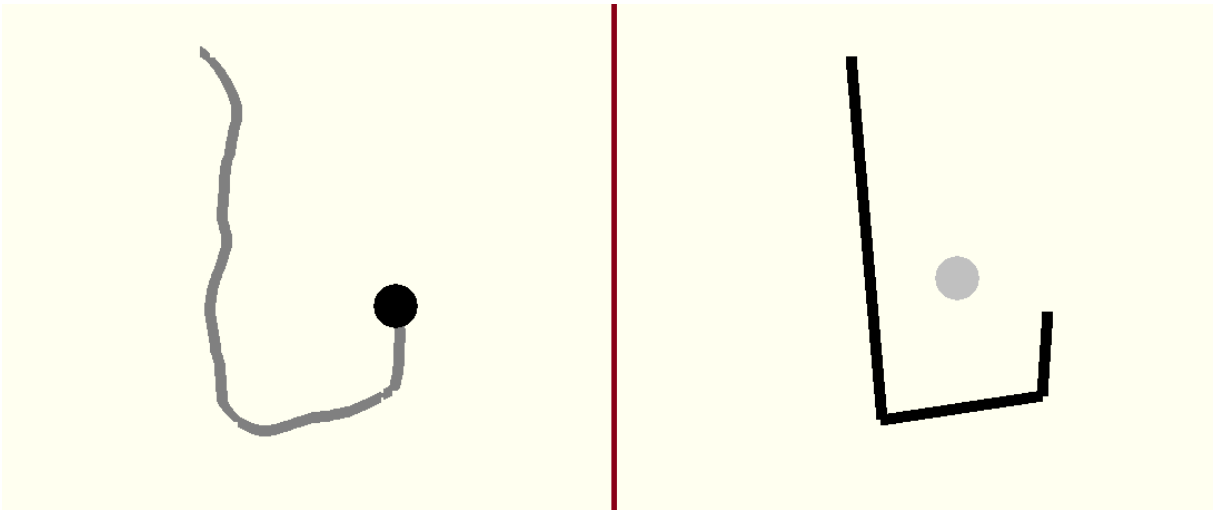

Figure 15. Stroke normalization with 4 feature point extraction

Example of the stroke normalization is shown in the Figure 15, where stroke on the left is unnormalized and the one on the right is after the needed feature points were extracted.

4.4.2 Character comparison

After the needed amount of strokes for the particular character have been drawn, the application does not let user to draw any more lines and waits to get the command to start comparing drawn character to the original stroke pattern.

Any lines that are considered to be wrong, are coloured red, the right ones are set green.

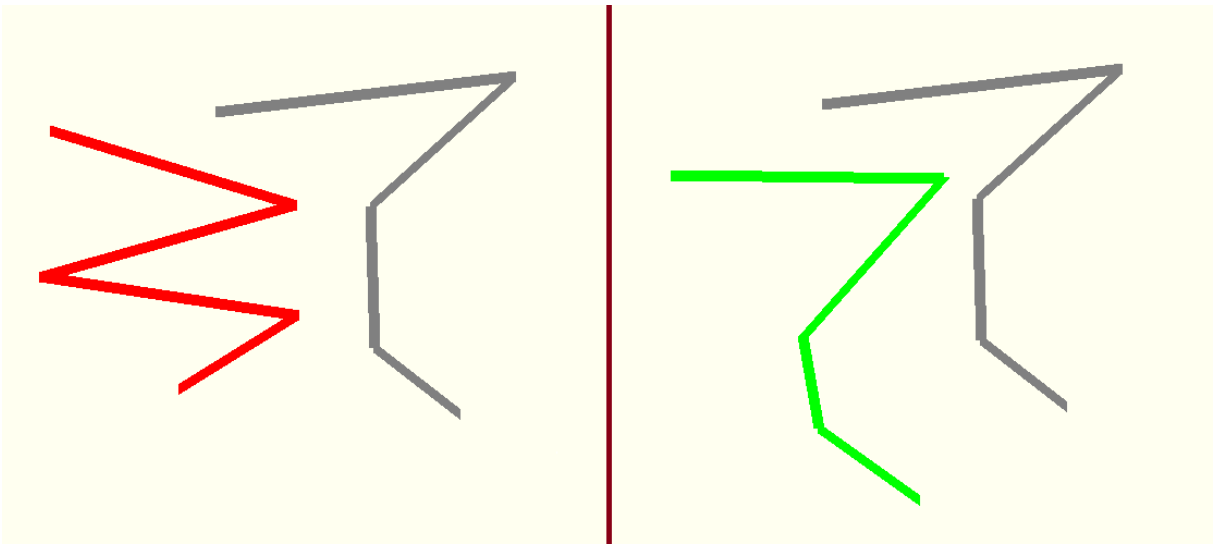

Figure 16. Display of drawn strokes after comparison

4.5 GUI

All of the GUI elements in this application were created using Pygame.

In order to use the Pygame modules, they first have to be initialized. The `pygame.init()` method initializes all of the main modules.

Main window is created using the `pygame.display` method. It allows creating the display *Surface* object easily, by just setting its size. *Display* is a very important *Surface* object, because all of the other *Surface* objects have to be drawn on it. To update the content of the entire display the `pygame.display.flip()` was used.

For framerate management `pygame.time.Clock()` object was used.

4.5.1 Screens

The application does not have many screens. The home screen is the first one user sees and the one that has the character widget on it, so the user can choose a character to learn.

After the character is chosen, application changes to drawing screen. To prevent any accidental strokes, after the screen change, it is not possible to draw. User has to wait till all the lines of the original stroke parent have appeared and only after that the drawing is possible.

4.6 Convenience

A few extra functionalities were created for convenience of the developer. They made writing and debugging code easier and faster.

4.6.1 Properties

Many variables were used in many modules of the application. For convenient editing reasons and to keep repetitions at minimum the `Properties` module was made. This module stores variables for application width and height, as well as font style and size etc. It also stores a few of the enumeration type variables used all over the application.

```
def enum(**enums):
    return type('Enum', (), enums)

Color = enum(SILVER = (192, 192, 192),
             GRAY = (128, 128, 128),
             BLACK = (0, 0, 0))

APP_WIDTH = 800
APP_HEIGHT = 600

FONT = "../../fonts/07LogoTypeGothic-Condense.ttf"
FONT_SIZE = 50
```

4.6.2 Logging

For practical reasons logging was added to the application. It allows for easier debugging and information gathering about the way the application is working. Logging entries display the date and time of the entry and the message like: `2015-05-15 17:20:58,448 Connected`.

```
logging.basicConfig(filename = 'Leap.log', level =
logging.DEBUG, format='%(asctime)s %(message)s')
```

5 Testing the application

Any developed application needs to be tested. So we ran a few tests to determine the work of the main functions.

First we tested pointer tracking and gesture recognition by Leap Motion. Then, the stroke detection method was tested. Lastly we gave the application for testing to four people, to see how intuitive the controls are.

5.1 Leap Motion pointer tracking testing

The pointer tracking is an important part of the application. So it was tested in different conditions, to determine the optimal environmental parameters.

Tests of different pen-like objects revealed, that Leap Motion works best with either fingers or traditionally shaped pens. Uneven shapes, holes and shining surfaces in objects hinder the performance of the unit. Objects under 5mm and over 20mm were hard to detect. Also brighter and duller coloured objects got better results.

By trying many *Pointables* at once, we got to the conclusion that Leap Motion works better if difference from the screen between the main *Pointable* and any other is at least 5cm.

Insufficient lighting also hinders the work of the device.

After a while, the device also seems to be overheating, and the performance of the device degrades. For better use of the device, giving it breaks to cool is advised.

5.2 Gesture testing

We also tested the gestures used in the stroke detection application.

Tests for all the gestures resulted in similar results. Leap Motion did not always recognize the gestures and they had to be repeated. Most difficulty was experienced in recognizing the *Key tap* gesture, which needed to be repeated several times for the device to recognize the gesture.

5.3 Stroke detection testing

For testing the stroke detection method we tested different *hiragana* characters.

Simpler *hiragana* characters, with smaller amount of points and curves were substantially easier to draw correctly because of the normalization. When the line consists of only two points, then the only ones that have to be drawn correctly are the start and the end point of the stroke.

More complicated characters are much more difficult. The normalization algorithm takes the most important points from the drawn ones for normalization. If they are too different from the ones from the original stroke pattern then the method determines them as wrong strokes. But this adds an extra level of difficulty and challenge to the learning task.

5.4 Intuitiveness testing

We wanted controlling the application to be intuitive and natural. So we decided to give the application for testing to four people, who had not seen or heard about it or its control system.

All four of the test subjects were able to understand the work of the pointer with ease. They said, that the drawing and clicking with it was logical and intuitive and they had no problems in understanding when pointer was in writing mode without prior instructions.

Gestures, on the other hand, needed some elaboration. Two out of four people were able to figure out the return to home screen gesture. After explaining the gestures and the activated actions, everyone found them easy to use.

6 Conclusion

We wanted to create a simple yet highly functional application for Leap Motion that would be practical and entertaining learning tool. To realize that goal we made tool for learning the basic writing method of Japanese *hiragana*.

As the main programming language Python was used for its simplicity and compatibility with other technologies. For creating mainly the UI elements with keyboard and mouse events the Pygame, that is a set of free Python modules designed for writing games, was used. For motion control, a small device equipped with optical sensors and infrared light capable of tracking and recognizing hands, fingers and pen-like tools, called Leap Motion was used. Its software is easy to use and provides a lot of different tools for managing the input from that particular device.

For the purpose of this application, the stroke detection method was developed. Stroke detection is a method that compares drawn strokes and original stroke patterns with each other and determines their compatibility. First, this method normalizes the strokes, by extracting the feature points from drawn strokes to match the amount in the same stroke from the original stroke pattern. Next it compares the gotten stroke with the original stroke pattern and decides if the strokes are deemed similar or not and returns the result.

In developing the application, a user interface was created. The control methods included the controls with Leap Motion pointer and gestures, but also the keyboard and mouse. In addition a great way to store the original stroke patterns was found – XML file. That enabled us to make character sheets easily changeable, that led to the idea of creating the adaptive application, that could potentially be used for learning any kind of character as long as the character consists of strokes and is stored in that particular way.

After the application was finished a few tests were ran, including the tests of Leap Motion tracking of the pointer and gestures, tests of the stroke detection and intuitiveness of the application itself. The test results were overall positive and satisfactory.

For future development, we would suggest working on the UI and adding new functionalities like quizzes and character sheet changing. Furthermore, better visuals for displaying the

correct way of writing the original characters could be implemented. Also, some improvements could be made for the comparison of the whole character comparison.

7 References

1. Leap Motion product. [WWW] https://www.leapmotion.com/product (18.05.2015)
2. Leap Motion overview. [WWW] https://developer.leapmotion.com/documentation/python/devguide/Leap_Overview.html (18.05.2015)
3. Leap Motion tracking model. [WWW] https://developer.leapmotion.com/documentation/python/devguide/Leap_Tracking.html (18.05.2015)
4. Leap Motion gestures. [WWW] https://developer.leapmotion.com/documentation/python/devguide/Leap_Gestures.html (18.05.2015)
5. Leap Motion coordinate mapping. [WWW] https://developer.leapmotion.com/documentation/python/devguide/Leap_Coordinate_Mapping.html (18.05.2015)
6. Pygame about. [WWW] http://www.pygame.org/wiki/about (19.05.2015)
7. About SDL. [WWW] http://www.libsdl.org/ (19.05.2015)
8. Pygame `pygame` module. [WWW] http://www.pygame.org/docs/ref/pygame.html (19.05.2015)
9. Pygame `surface` module. [WWW] http://www.pygame.org/docs/ref/surface.html (19.05.2015)
10. Pygame `display` module. [WWW] http://www.pygame.org/docs/ref/display.html (19.05.2015)
11. Pygame `font` module. [WWW] http://www.pygame.org/docs/ref/font.html (19.05.2015)
12. Pygame `draw` module. [WWW] http://www.pygame.org/docs/ref/draw.html (19.05.2015)
13. Pygame `event` module. [WWW] http://www.pygame.org/docs/ref/event.html (19.05.2015)
14. Pygame `key` module. [WWW] http://www.pygame.org/docs/ref/key.html (19.05.2015)
15. Pygame `mouse` module. [WWW] http://www.pygame.org/docs/ref/mouse.html (19.05.2015)

16. Pygame `time` module. [WWW] http://www.pygame.org/docs/ref/time.html (19.05.2015)
17. *Hiragana*. [WWW] http://en.wikipedia.org/wiki/Hiragana (20.05.2015)
18. Bilan Zhu and Masaki Nakagawa (2012). Online Handwritten Chinese/Japanese Character Recognition, Advances in Character Recognition, Prof. Xiaoqing Ding (Ed.), ISBN: 978-953-51-0823-8, InTech, DOI: 10.5772/51474. Available from: http://www.intechopen.com/books/advances-in-character-recognition/online-handwritten-chinese-japanese-character-recognition
19. Projection of point onto line 2D. [WWW] http://intumath.org/Math/Geometry/Analytic%20Geometry/projectionofpoin.html (21.05.2015)
20. Tegaki-zinnia-jaanese-0.3.zip. [WWW] http://www.tegaki.org/releases/0.3/models/tegaki-zinnia-japanese-0.3.zip (22.05.2015)
21. GNU Lesser General Public License, version 2.1. [WWW] https://www.gnu.org/licenses/lgpl-2.1.html (22.05.2015)
22. Leap Motion pointable. [WWW] https://developer.leapmotion.com/documentation/python/api/Leap.Pointable.html (23.05.2015)